The Dragons Arena System
Distributed Simulation of Virtual Worlds

Distributed Computing Systems IN4391

Georgios Oikonomou 4330544 g.oikonomou@student.tudelft.nl

Michail Vasilakis 4314743 m.vasilakis@student.tudelft.nl

Course Instructor: Alexandru Iosup

Course Assistant: Yong Guo

Abstract

Online gaming is a very challenging field in distributed computing because of its consistency, scalability and fault tolerance issues. A large part of the ongoing academic research aims to tackle the above problems. The WantGame Company in order to improve its provided products and follow the emerging cutting-edge technologies, has decided to switch the Dragons Area System functionality to the distributed world. The company provides a single server implementation of a virtual world game which has to be modified both in architecture and implementation. In this article, a distributed fault tolerant version of that game is presented along with the results of simulation experiments evaluating the game's scalability, fault tolerance and performance.

1. Introduction

Nowadays, computer games are considered one of the most popular ways of entertainment. More specifically, online games attract more and more users every day which has made the massively online games a real challenge. The main characteristic of massively online gaming is that millions of users can use their internet connection to play multiplayer games, interacting with one another and cooperating. This implies that mechanisms are required for dealing with such great communication cost and computing workload. Our project regards the design and implementation of a distributed system for an online game named Dragon Area System (DAS) which supports many concurrent users interacting in a simple virtual world.

For the current project an existing implementation of the game was given which is based on a Client/Server implementation. This means that there is a centralized component which handles all the requests by the users and keeps track of the games state. This implementation might be simple but it has a single point of failure. If the main server suffers from any type of failure such as crashing or omitting the receptions then the game stops for all players. This single point of failure, has also implications in scalability, resulting in limiting the amount of users that a system can serve during the game.

Our main objective is to develop a distributed version of the aforementioned game based on the features that the game offers. The game will be hosted in a distributed system including more than one active servers which will share the same game. The main requirements of the system that should be taken into consideration are the consistency of the game, its fault tolerance and scalability. The system will also provide a simple way of load balancing and communication efficiency.

The rest of the article is structured as follows. In Chapter 2, the DAS game and its requirements are explained. In Chapter 3, the new system's design is described along with the implementation of the features that the WantGame Company has required. In Chapter 4, the results of the experiments are presented and explained. Finally, Chapter 5 and 6 refer to our conclusions.

2. Application Background

The Dragons Area System application is a simple warfare game between computer-controller dragons and hundreds of virtual knights (avatars or real-life humans). The game's avatars are the player-knights and the dragons. The game is played in a virtual world where each player controls a knight. The virtual world is represented with a grid battlefield where each avatar occupies a single square and two avatars cannot occupy the same square. All the avatars have health points which can be used to indicate when an avatar is alive or dead. In addition, each avatar has attack points that can be used to interact with other avatars. Each player can perform a move action in which they are moved 1 step horizontally or vertically but not both at the same action. The dragons are steady and cannot move. A player can strike dragons close to him and vice versa which results in subtracting the attacker's attack points from the victim's remaining health points. Players can also heal other nearby players by adding to the receiver's remaining health points the healer's attack points.

The consistency feature of the DAS game demands the server sharing the same game state in every connected player and the game actions should be issued and played in the correct chronological order. If the server delays on handling an action, a game rollback should be introduced to a previous valid checkpoint.

The given DAS game does not provide fault tolerant features due to the operation of the single server implementation. The distributed version of the DAS game should operate properly even in the event of a component's failure. The shared game among the different servers will proceed even if multiple servers crash. The connected players of a malfunctioned server will be removed from the game when the failure is identified by other servers.

With regard to scalability, by having more active servers means that the game should still be consistent as well as capable to serve more clients in total. Finally, the performance of the game lies on the latency among servers which have to communicate efficiently to achieve real-time interactions.

3. System Design

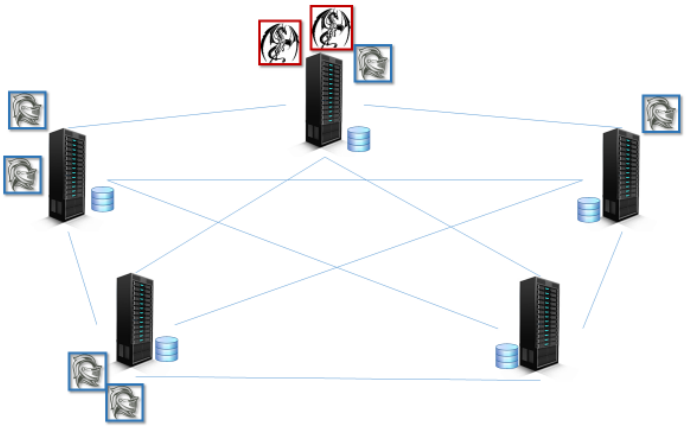

Figure 1: System Architecture

System Architecture

The implemented distributed system consists of a decentralized group of servers that are collaborative and communicative (mirrored servers). Every server can communicate with the rest of the servers in the group and can have a limited amount of connected clients. Each client chooses to connect to the closest server according to its geographic position in order to reduce the communication latency between them. The game state is logged in all the servers in order to provide reliability with multiple game replicas. The system includes a server with an additional responsibility to operate the automated game entities (dragons). The system architecture is depicted in Figure 1.

The communication functionality is implemented using the Java Remote Method Invocation (Java RMI) API for sending messages between the system components. Thread processes are also introduced to provide parallel execution of independent tasks. For the game simulation client's behavior is replaced with computer-controlled commands that emulate human behavior according to the proposed strategy. It is assumed that all the servers share the same clock.

Consistency

Game consistency among the servers is maintained by keeping a local copy of the game in every server. Each server keeps two lists with pending and valid actions of the game including the timestamp that they were issued by the server. The main consistency mechanism of servers considers an incoming game action of a client to be issued as a pending action before becoming valid. For an action to become valid from a server, the server has to verify that there was no other conflicting action that should have been issued earlier. A conflicting action is defined as a logged pending action that belongs to another server and was issued earlier.

In order to achieve that, the server asks the other servers, running the game, whether a pending conflicting action is logged to their lists. In order to avoid communication overhead, each server replies back only if he argues with the action. If there are no replies, the pending action becomes valid after a specified timeout. This timeout is supposed to be twice the value of the

maximum time a message needs to be sent to any server. Any conflicting pending action is removed from the corresponding pending log in the system.

Whenever an action is validated then it is instantly broadcast to all the servers in order to update their valid logs. Having this rule it is guaranteed that the actions issued in different servers are logged and played by every server in the same order. An additional benefit of such a mechanism is the maintenance of multiple logging replicas which can be vital in case a server crashes while the rest of the servers can continue running the game.

The first server creating the game controls the dragons' actions and notifies the rest of the server for their new moves.

A checkpoint of the current game's state is saved periodically in every server and when an inconsistency occurs then the local valid checkpoint updates the game states of all servers.

Fault tolerance

The main mechanism to identify failures in the client or server side is sending periodic messages (ping messages) to notify the component's state to others.

If a client has failed the server starts a procedure to check the client's state. After a timeout period without a reply from the client, the server removes the corresponding game entity and broadcasts the update.

If a server has failed, the following procedure takes place. If server A has failed and it is noticed by server B then server B has to request from the rest servers the last state of server A. Every server that replies, he also broadcasts the reply to all servers. When server B receives all the replies then it decides to remove server A from the list of running servers. The same removal should be performed by any server who also receives all the replies. This mechanism supports a multiple failure scenario where the remaining servers decide upon the removal of the faulty servers.

If the server handling the dragons fails, then the above procedure is performed and the next server according to the servers' order, controls the next actions of the dragons.

Load balancing

Firstly, it has to be assumed that a client decides to connect to its closest (distance) server. During simulation experiments, this functionality is accomplished by a random selection of the server to connect. Every server has a limited number of clients that can serve. When the server has reached this limit, the new client will be redirected to the closest server with the least workload.

The game is created by the first server which cannot find any other running servers. By the time a new server starts operating, it checks whether there are other servers online. The game is not played yet in that server until the first client connects to it. This mechanism is preferred because it avoids useless communication among the servers as well as limits the usage of the available computing resources.

The project's implementation was published in GitHub repositories as suggested in the project's description and can be found in the following link

Project link: https://github.com/mikevas89/Dragonball

4. Experimental Results

Three experiments were performed in order to evaluate the implemented version of the game, with respect to scalability/performance, fault tolerance and scalability/load balance. Two local machines were utilized for the experiments running Windows and Linux operating systems correspondingly and connected to LAN.

The evaluation of the implemented system was mainly conducted by obtaining execution outputs that appeared when important system events happened such as a registration of a new client. For the collection of the results, message collectors were used along with parsing tools that were classifying the different types of the messages. These tools were also implemented during the system's development.

4.1 Experiment 1 - Scalability/Performance

For the evaluation of the performance of the system, the response time of a player's action was measured. By response time, we define the average time between an action is issued to a server until the other servers are notified for this new game action. This metric will present an estimation of a delay of an action to be played due to the load of servers with many clients. The figure below depicts the aforementioned response time over the total number of connected clients to the network.

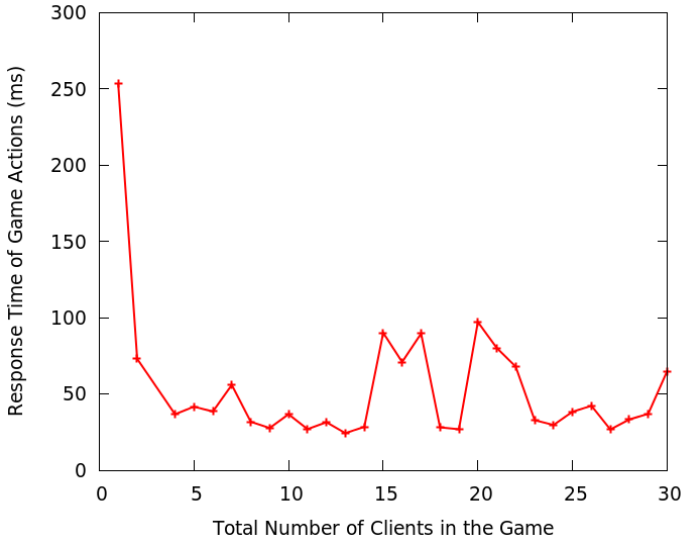

Figure 2: Average Response Time

As it can be seen, the average time seems to be approximately the same even with the increase of the clients. The slight fluctuations are probably caused by the irregular rate of incoming clients.

This result presents a beneficial property of the system which is able to handle more clients compared to the single architecture approach without considerably lacking in performance.

4.2 Experiment 2 – Fault tolerance

The purpose of the second experiment is to identify the mechanism of the system when server failures are introduced. A simulation of this includes the failure-stop of a server and the proper operation of the rest of the network. In the current experiment, two servers are hosting the game and each server holds around five clients. At some point, server B is stopped and after a while restarts.

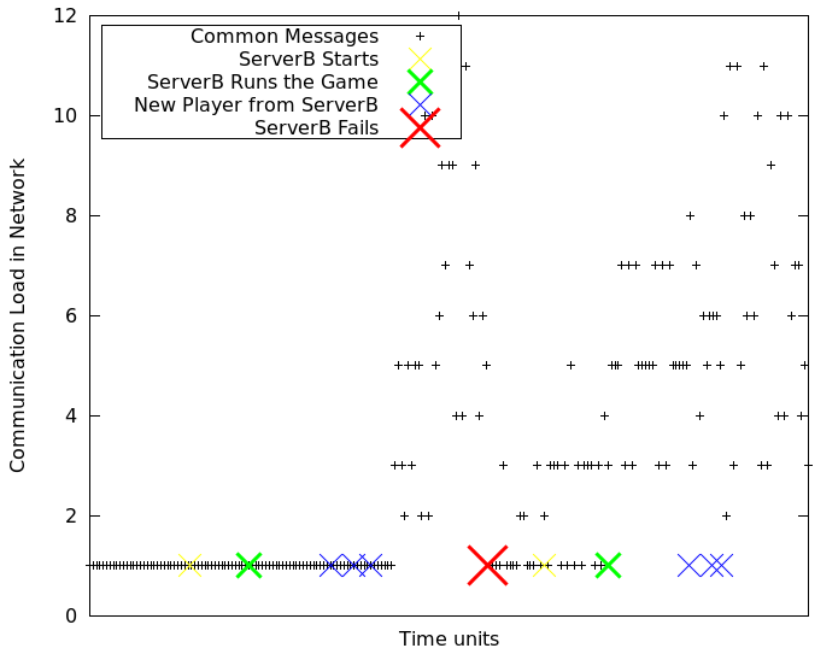

Figure 3: Behavior of network with Faulty Server

As it is depicted, server B crashes and the communication load is decreased because its clients are removed. By the time it fails, the other server still hosts the game properly but with less communication. The first server senses that the second server is down and removes its entities for the game. When the second server restarts, it recovers the connection with the alive server and only when a new client is connected, the running game state (from first server) is obtained by B. Eventually, the game is being played on both servers again.

4.3 Experiment 3 – Scalability/Load balance

Load balancing is a crucial feature for the distributed system. It is considered that clients connect to the closest (distance) server to avoid communication delays. This implies to test scenarios where many clients try to connect to specific servers. The figure below presents a scenario where clients are redirected to other servers to maintain load balancing. Additionally, another useful feature of the new DAS system is that servers with no connected clients are idle and not running the game. This results in a more efficient resource allocation.

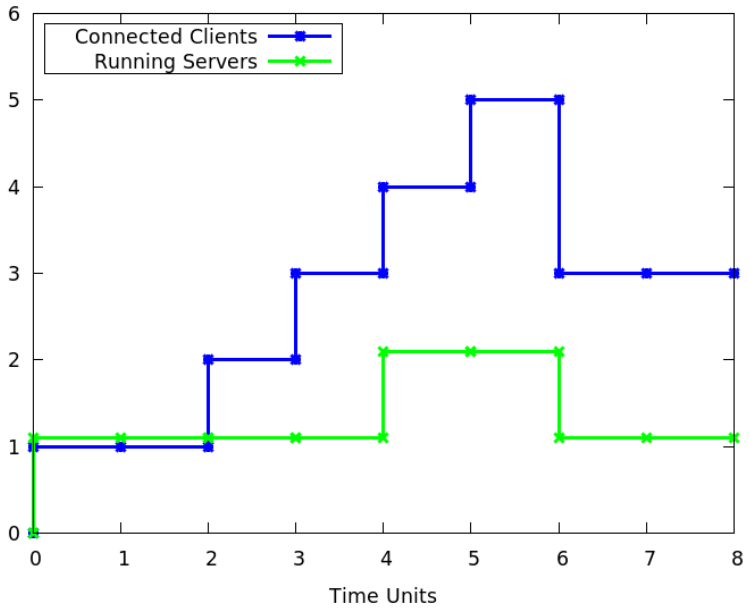

Figure 4: Load Balance of Server Load

In Figure 4 clients are redirected to second server when the limit of the connected client is reached. When the clients of the second server are removed from the game, the second server becomes idle but alive (no game communication) and the game is played only in the first server.

5. Discussion

A distributed implementation of DAS game has shown that some advantages are offered compared to single server approach. The results of the experiments presented above depict an improvement of the features of the previous implementation for the performance and the fault-tolerance of the network.

Tested cases where a server can crash and then re-enter the network without the game being aborted have been conducted to experiments. The results prove a stable network environment

for hosting a game. This is considered as a huge improvement since in a single server implementation a crashing server and continuation of game are not realizable.

The feature of consistency has also to be maintained using communication among the servers. This introduces a bigger communication overhead compared to having only one server organizing the whole functionality. This is a significant drawback of a distributed system's implementation but can be handled in mechanisms such the alive servers which are not hosting the game when there are no connected clients. It has to be mentioned though that if the game had a smaller latency like a First Person Shooting game, then the communication overhead would have been greater.

Communication overhead is also reduced with the mechanism of the consistent actions where servers reply only when an invalid action was occurred. A first impression of that reduction was the results of some experiments implementing full communication among servers. The communication load was too high for the exchange of non-critical messages which emerged the scalability incompetence of that DAS version.

From our point of view, WantGame Company should consider moving the DAS game to the distributed world because the advantages of such an implementation would eliminate the disadvantages. However it has to be mentioned that there might be cases where specific mechanisms that handle huge workloads may be also implemented when it comes to server millions of clients concurrently. This cannot be tested extensively in the context of this effort but having this kind of approach results in a more robust system which can handle a lot more load than the previous single-server option.

6. Conclusion

For the lab assignment of DCS, a new distributed system was designed given a single server implementation of an application. The DAS game was redesigned so that it can host more clients, handle possible node failures and still providing good performance. During the implementation, many challenges were found and had to be overcome with regard to consistency and fault tolerance of DAS. For the evaluation of the system, experiments were conducted in order to check the important system features of a distributed implementation. The new DAS is capable to handle cases where many clients are hosted, have problematic servers as well as share the workload in rush hours of computations. From the performed experiments, it can be concluded that the system meets most of its decided requirements and offers features that the single server approach could not have by default. It has to be mentioned that experimental could have been more accurate in more realistic distributed environments.

7. Appendix A: Time sheets

Bellow we present the time that it was spent to deliver the current project.

Task	Time
Total-time	Approximately 260 hours
Think-time	First two weeks of the course
Dev-time	Approximately 190 hours
Xp-time	Approximately 10 hours
Analysis-time	Approximately 3 hours
Write-time	Approximately 8 hours
Wasted-time	Approximately 15 hours